SciVerse ScienceDirect

Procedia - Social and Behavioral Sciences 27 (2011) 160 – 168

Procedia
Social and Behavioral Sciences

Pacific Association For Computational Linguistics (PACLING 2011)

Using Language-Based Search in Mining Large Software Repositories

Normi Sham Awang Abu Bakar[a]*

[a]*International Islamic University Malaysia, Jalan Gombak, 53100 Kuala Lumpur, Malaysia*

Abstract

Language component plays an important role in data/information retrieval. Data retrieval in software engineering is often hindered by the difficulty of getting data from commercial software. The emergence of the open source repositories has contributed tremendously in the collection of software data. This paper highlights the data retrieval method for mining software from a vast open source software repository, SourceForge. For the purpose of automating the data retrieval from the repository, a parser was written using the Python programming language, and based on the pattern matching algorithm. The retrieved data were later used to estimate the quality of the open source software.

Keywords- Data retrieval; Software repository; Language – based search; Automation; Software quality

1. Introduction

The application of language-based search can be extended to basically any field. Essentially, when there's a need to search for any topic, several types of search keywords can be used to find a match in the general text or database. One of the applications of language-based search is in the mining of empirical software engineering data.

Empirical research seeks to explore, describe and explain natural, social or cognitive phenomena by using evidence based on observation or experience. It involves obtaining and interpreting evidence by experimentation, systematic observation, interviews or surveys, or by careful examination of documents or artefacts.

An empirical body of evidence in empirical software engineering can be described as a set of studies, each performed under certain explicit conditions, for which both quantitative and qualitative, subjective

* Corresponding author.
E-mail address: *nsham@iium.edu.my*

doi:10.1016/j.sbspro.2011.10.594

and objective data have been collected and based on which certain conclusions and interpretations have been provided [3]. Empirical research makes an extensive use of evidences based on data, thus, data gathering plays an important role in this research.

Collecting software engineering data is often a challenge due to the scarcity of data especially when dealing with commercial software. Generally, research conducted on commercial product development is obstructed by restricted access to the development process and selectively released data. Companies that commercialize their software products are in most cases not interested in sharing the product's source code due to the risk of code spilling over to competitors or "software pirates". On the contrary, due to their development practices, open source software projects show considerably high transparency of data for research. The software's source code is generally available from repositories that host the project, for instance, SourceForge and Freshmeat, as well as from the websites of the open source projects themselves, such as Apache [4], Mozilla [5] and OpenBSD [6].

The emergence of the open source software development paradigm has stirred interest among software developers and everyday software users in terms of how software is developed and also in terms of general usage of the software. According to the definition given by Open Source Initiative [7], open source software (OSS) allows users to have access to the source code of the software, the freedom to use the software as they see fit, modify the software to create derived work, and redistribute the derivative software for free or at a charge. The users of the software could modify or use the software according to their own needs.

Furthermore, most projects host mailing lists dedicated to various aspects of the software product and the project. Some lists may focus on technical development issues, while others may deal with user assistance, general user feedback, or discussions regarding the "philosophy" of the project [8]. These lists represent exceedingly valuable data for researchers because the make discussions available that can be used to examine multiple issues in open source development. The retrieval of both technical data and mailing lists data is in many cases possible for all researchers, and not only restricted to those who have exclusive relationships with developers. For many open source projects, such data are extensive, covering millions of lines of code and thousands of messages and, therefore, are useful for various forms of quantitative analysis.

This paper proposes a methodology that automates retrieval of public data from a project hosting site, SourceForge.net [9], spanning a large number of open source projects both large and small, and using information stored by the available software development and communication tools.

2. Background

The objective of collecting the data from an open source repository is to collect several information pertaining to software quality measures which include user reported defects, software complexity, number of downloads, and number of developers.

Initially, the systems to be used in this research were chosen manually. However, the manual selection process was very time consuming, especially when the repository stores more than 300,000 systems.

The projects in SourceForge were grouped into several categories based on functionality, such as communications, database, education, games, internet, multimedia, office/business, security, software development and many more. For the purpose of this research, 104 projects from four different categories were chosen, namely:

- Games/Multimedia

- Internet/Communication
- Office/Programming/Database
- Scientific/Engineering/Operating Systems

Only systems which fulfill the following criteria were selected in this research:

1. Active projects (Activity percentile more than 90 percent) – SourceForge ranking based on the overall project activity (combination of project traffic, development and communication).
2. Developed using Java programming language.
3. Development status is : Production/Stable or Mature.
4. High number of total downloads (number of downloads more than 50,000)
5. Availability of error reports (error reports shown in bug tracking system).

In order to ease the process of searching the systems which meet all the criteria, a parser was written using Python programming language to automatically search and download the relevant systems. The parser has shortened the data collection process of this research compared to the manual collection method and thus, more effort can be given to the data analysis process. Researchers can benefit tremendously when using such parsers since more substantial effort and resources can be allocated to data analysis and model development.

3. Mining Software Repositories

Software repositories contain a wealth of valuable information about software projects. Examples of software repositories are [10]:

- Historical repositories such as source control repositories, bug repositories, and archived communications record several information about the evolution and progress of a project.
- Run-time repositories such as deployment logs contain information about the execution and the usage of an application at a single or multiple deployment sites.
- Code repositories such as Sourceforge.net and Google code contain the source code of various applications developed by several developers.

In this research, the code repository was used as the source of data for quality measurement. The repository, SourceForge, is the biggest open source repository and one of the main open source repositories. Hence, it is essential to ensure that the data retrieval can be automated to save a significant amount of time as well as to allow replication by other researchers.

The downloading process started with running the Python script to search for the relevant systems based on the specified criteria. Later, these systems were automatically downloaded from the repository and were kept in a database to be analyzed later.

The next step was to extract the metrics/measurement to investigate the quality level of the open source software. The metrics data then were stored in a metrics database for analysis and analysis purpose. The data retrieval process is summarized in Figure 1.

The project retrieval and parsing process started with executing the Python script which was written to search for relevant systems based on certain keywords or criteria given in Section 2. Once the parser managed to locate the systems which match all search criteria, the systems were downloaded and stored in a repository/database. The search process applied a pattern matching algorithm which will be discussed in detail in Section 4. The example of the Python parser/script is shown in Figure 2.

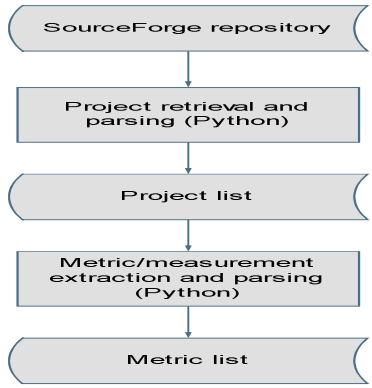

Fig. 1. Data retrieval process

```
## check one - download >= 50000
dvalue = 0
for i in download:
     if i >= '0' and i <= '9':
          dvalue *= 10
          dvalue += int(i)
     if( dvalue < 50000 ):
          continue

## check two - good developing status
       status1 = re.search('5 - Production/Stable', dev )
       status2 = re.search('6 - Mature', dev )
          ok = 0
       if( status1 ):
          ok = 1
       if( status2 ):
          ok = 1
       if( ok == 0 ):
          continue

## check three - language java
       if( 'Java' != xlang ):
          ok = 0;
       if( ok == 0 ):
          continue
     out.write( '<h1>' + name + '</h1>')
     out.write( 'Total Download ' + download )
     out.write( dev )
     out.write( lang )
```

Fig. 2. Example of the Python parser

4. Pattern Matching Algorithm

In order to write the parser to retrieve the data from the extremely large repository, a pattern matching algorithm was used. Essentially, there are two types of pattern matching algorithm normally used: exact and inexact pattern matching. Exact pattern matching involves finding all occurrences of a pattern *P* in a string *S*, where *S* is longer than *P*. Being the simplest form of pattern matching, exact pattern matching is still widely used in a variety of text searches from internet search engines to word processing. With inexact matching, the concern shifts to alignment of sequences based on their similarity. The nature of the problems of sequence alignment and the numerous routes one could take to address them yield a variety of categories of alignment methods, ranging from pairs of sequences to large groups [11].

This paper emphasizes the use of exact pattern matching to retrieve data from the selected repository, SourceForge. While higher processor speeds and other advances have reduced search response to negligible times, exact matching still remains a useful area of study and development for a number of reasons. First, as information continues to grow, sequence searches will become increasingly taxing on search engines. Searching for patterns in sequences will invariably be more difficult than performing common internet searches because of fewer unique constraints, and the absence of spaces denoting words/patterns.

Secondly, the exact pattern match still remains an integral part of faster matching algorithms, typically comprising the final part of a search. Lastly, an understanding of the classical methods of exact pattern matching lends itself to the development of new algorithms.

4.1 Naïve method

The simplest method of exact matching is the naïve method. The premise is simple: the first letter of pattern *P* is lined up with the first letter of *S*, and the letters of the aligned region are compared until all of *P* is found to match the corresponding *S* region or a mismatched letter is found, in which case *P* is shifted one letter to the right and the process is repeated [11].

Example:
Development status: **Stable**
Stable
Stable
..
.. **Stable**

The apparent disadvantage of this method is the speed of the search technique. By moving one character at a time, the worst-case time required to compare the pattern to the entire sequence would be proportional to the length of *P* multiplied by the length of *S*.

Initially, this method seemed to be suitable to be used in this research. However, because of the performance problem, another method, the Boyer-Moore algorithm was considered for the data retrieval.

4.2. Boyer-Moore algorithm

The Boyer-Moore algorithm [12] provides marked improvement over the naïve method through the implementation of three key ideas. First, although the pattern *P* and string *S* are aligned on the left, the algorithm scans for matches from right to left. Secondly, when a mismatch is found between letter *p* in P and letter *s* in *S*, P is moved to the right so that the rightmost occurrence of *s* in P is aligned with *s*; if *s* does not occur in P, P is shifted one space to the right of *s* (“Bad Character Shift Rule”).

Example:
Programming language: Java

Java	1. ‘a’ – ‘g’ mismatch
Java	2. ‘a’ in “Java” shifted to ‘a’ in Programming”: ‘a’ – ‘m’ mismatch
Java	3. ‘a’ is shifted to ‘a’ in “language”: ‘a’ – ‘g’ mismatch
Java	4. Lastly, ‘a’ is shifted to ‘a’ in “Java”: match found

Lastly, the algorithm implements a third rule which is improved upon by Gusfield [13]. The modified rule ("Strong Good Suffix Rule") states that if the suffix Q of pattern *P* matches a substring *R* in *S*, when a mismatch is found, P is shifted to the right so that Q' is matched up with R, where Q' is defined as the rightmost occurrence of Q in P with a different letter to the left. If *Q'* does not exist then the left end of *P* is shifted past the left end of *R* until another match is found, or until *P* is shifted past *R*.

Example:
Development status: Stab**le**
Stab**le** 1. Suffix "le" matches and the whole word matches

The combination of the three characteristics of the Boyer-Moore algorithm makes the method simple yet powerful. Immediately, searching from right to left with the Bad Character Shift Rule, the potential to skip over a greater number of non-matching substrings is realized.

5. Results Discussion

Once all systems have been retrieved from the repository, the metrics extraction begins. The objective of this research is to see the general correlation between several metrics/measures, such as, structural complexity, class size, number of downloads and number of developers with system defects as the dependent variable. Most often, software quality is measured by the number of defects that occur in the system and the user-reported defects were collected from the open source repository using a Python parser. A software product is considered defective when it does not perform its functions according to the user's expectations [14].

The retrieval process returned 104 open source systems from the four specified categories which meet all the given criteria. Later, the relevant metrics were extracted from the retrieved systems, and analyzed using statistical analysis. Each independent variable is compared to the user reported defects to investigate which independent variable influence defects, which indicates the quality level of the systems.

The total defect counts then need to be normalized by the system size. In this research, the effective lines of code (eLOC) is used as the size measure. This measure is then divided by 1000 to derive another measure, KLOC, which is later used in the formulation of Defect Density (DD):

$$DD = \frac{\sum \text{Defects}}{\text{KLOC}}$$

Defects = Total defects collected
KLOC = The number of thousands (1000) of lines of code in system

The results obtained in this research show that only structural complexity has influence on defects, with p-value = 0.001 and Spearman correlation = -0.33. The scatter plot of the relationship between these variables is illustrated in Figure 3. The figure shows that the majority of the systems have defect density of less than 2 which is very good. This means, for every 1000 lines of code (LOC), only 2 defects occur.

Besides the structural complexity, another independent variable studied is the number of downloads. The reason to include this measurement in this research is to see whether downloads influence the user reported defects, since open source systems rely on the defects reported by the users to improve their quality. The results show that the number of downloads is not correlated with defects with p-value = 0.27 and Spearman correlation = -0.02. The relationship between these variables is depicted in Figure 4.

Finally, the number of developers is another independent variable used to estimate defects. The results show that it is not significant is estimating defects (p-value = 0.47, Spearman correlation = -0.07). The scatter plot in Figure 5 shows that the majority of the systems have less than 40 developers.

Whilst the number of developers in each project might not be a good indicator of the number of defects reported by users, there is a possibility that developers' experience does have an effect on defects. However, it is not possible to obtain this information in this work due to the nature of the systems being investigated, i.e. open source systems, where it is difficult to collect the data concerning the experience of all developers in the systems.

Most developers who are involved in the development of open source projects are experienced and skilled people who are professionals in the technology industry, as state by O'Neil [15]:

"A statistical study by Lakhani and Wolf [16], which surveyed developers from a random sample of open source projects on Sourceforge.net, found that a solid majority of contributors were experienced, skilled individuals with jobs in the technology industry. The average contributor had more than a decade of programming experience; 55 per cent worked on open source projects as part of their job."

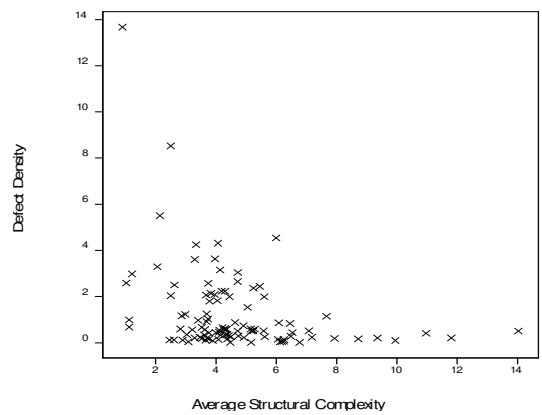

Fig. 3. Scatter plot of structural complexity and defect density

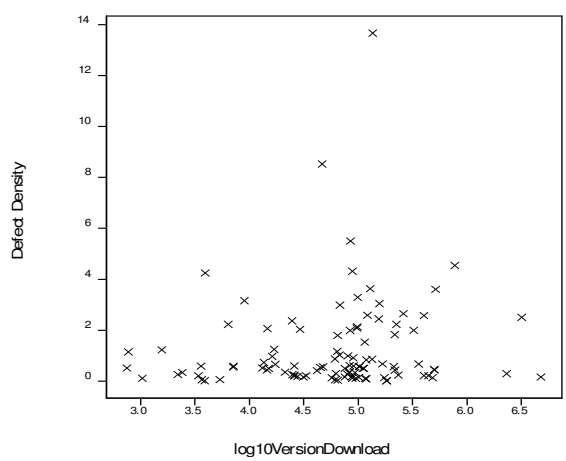

Fig. 4. Scatter plot of the number of downloads and defect density

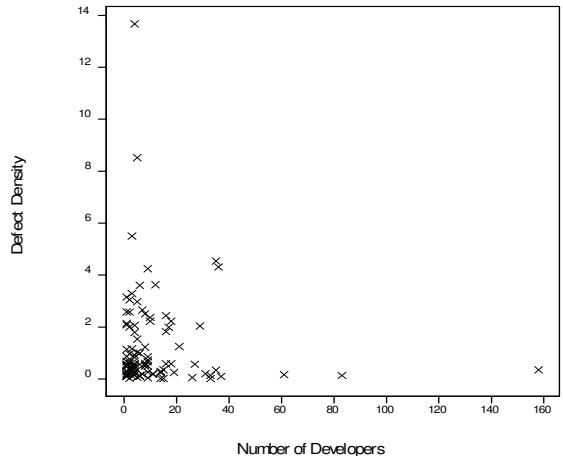

Fig. 5. Scatter plot of the number of developers and defect density

6. Conclusion and Future Work

The application of the data mining techniques is spanning across many fields of research. In the empirical software engineering area, mining software data from large software repositories has been the centre of many research studies. The emergence of various open source repositories has made the software data more accessible to researchers. However, going through the vast amount of data can be overwhelming if the retrieval is done manually.

This paper presents the methods used to retrieve data automatically from a massive open source repository, SourceForge. A parser was written in Python programming language to retrieve the related systems based on several search criteria. The pattern matching algorithm was used to find the right systems to be included in this research.

The automated data retrieval helps to save a significant amount of time in the data collection process. Hence, the researchers can put more effort in analyzing the data and possibly come out more meaningful findings.

However, there are some limitations to the parsing technique. The search can be quite slow when the parser is applied to extremely large repositories like SourceForge. Another limitation for this parser is, it can only perform the search in one repository at a time, for example if a researcher needs to conduct the search in several repositories, the person needs to run the parser separately. In other words, the parser cannot perform the search in multiple repositories simultaneously.

In reference to the previously stated limitations, the parser can be improved by using a more efficient algorithm, and that is the plan for the future. Another possible improvement is to write a parser which can retrieve data from multiple repositories simultaneously.

References

[1] Harrison, R., N. Badoo, E. Barry, S. Biffl, A. Parra, B. Winter, and J. Wuest, Directions and Methodologies for Empirical Software Engineering Research, in Empirical Software Engineering, vol. 4(4), pp. 405–410, 1999.
[2] Lehman, M. M., and L. A. Belady, A Model of Large System Development, in IBM Systems Journal, vol. 15(3), pp. 225–252, 1976.
[3] Briand, L. C., Empirical Evaluation in Software Engineering: Role, Strategy and Limitations, in Empirical Software Engineering Issues, vol. LNCS 4336, p. 21, Springer-Verlag, Berlin, Heidelberg, 2007.
[4] www.apache.org
[5] www.mozilla.org
[6] www.openbsd.org
[7] www.opensource.org
[8] von Krogh, G., and S. Spaeth, The Open Source Software Phenomenon: Characteristics that Promote Research, in Journal of Strategic Information Systems, vol. 16(2007), pp. 236–253, 2007.
[9] www.sourceforge.net
[10] Hassan, A. E., The Road Ahead for Mining Software Repositories, FoSM, pp. 48-57, 2008.
[11] Lee, J., Analysis of the Fundamental Exact and Inexact Pattern Matching Algorithms, FoSM, pp. 48-57, 2008.
[12] Boyer R.S. and Moore J.S. A Fast String Searching Algorithm. Comm. ACM, 20:762-72, 1977.
[13] Gusfield D., Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology, 1997.
[14] Conte, S. D., H. E. Dunsmore, and V. Y. Shen, Software Engineering Metrics and Models, Benjamin/Cummings Publishing, California, 1986.
[15] O'Neil, M., Cyberchiefs, 61 pp., Pluto Press, 2009.
[16] Lakhani, K. R., and R. G. Wolf, Perspective on Free and Open Source Software, chap. Why Hackers Do What They Do: Understanding Motivation and Effort in Free/Open Source Software Projects, pp. 3–21, MIT Press, 2005.